Technische Universiteit
Eindhoven
University of Technology

Department of Electrical Engineering

Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
http://w3.ele.tue.nl/nl/

Series title:
Master graduation paper,
Electrical Engineering

Commissioned by Professor:
Prof. Dr. H. Corporaal

Group / Chair:
Electronic Systems

Date of final presentation:
February 09, 2012

Report number:

Saliency Detection on FPGA Using Accelerators and Evaluation of Algorithmic Skeletons

by

Author: S.H. Wang

Internal supervisors:
Prof. Dr. H. Corporaal
Ir. Z. Ye

Saliency Detection on FPGA Using Accelerators and Evaluation of Algorithmic Skeletons

S. H. Wang
Department of Electrical Engineering, Electronic Systems Group
Eindhoven University of Technology
Email: s.h.wang@student.tue.nl

***Abstract*—Real-time vision applications are becoming more interesting to use as more computing power is available, but often those applications are still too compute intensive. Therefore, an FPGA can be used to accelerate the application. However, making an FPGA design is a time-consuming task and many errors can be made. In this paper, we use a mixed design approach of manually designed accelerators and accelerators designed using a high level synthesis tool, because some functions are not supported in the high level synthesis tool or are not as efficient as a manual designed accelerator. Further, we analyzed the possibility of using skeletons for certain classes of functions with a high level synthesis tool. By using skeletons the programmer does not need to have knowledge of the whole architecture, while efficient designs can still be made. A saliency detection application is used as a case study. For this application, a speed-up of 3 times is achieved compared to an Intel Core i5 running at 2.53 GHz. Design time is greatly reduced by using a high level synthesis tool. The manually designed accelerators took a few weeks to complete, but using the high level synthesis tool only a few days were needed. In this case study, the high level synthesis tool shows to be promising to use for skeletons.**

I. Introduction

One of the main purposes of technology is to assist men with our daily job and activities. For example, the personal computer and the internet has brought many conveniences, such as long range video chatting and e-mail. One interesting domain of applications is the domain of vision applications. Vision applications can have many benefits as an aid. Some applications can detect and recognize objects such as speed signs in an image from a camera feed. With this information, the user can decide to increase or decrease his speed.

In this paper, we use the vision application which detects salient objects. Salient objects are objects that stand out relative to their neighborhood. Saliency detection is mostly used as an attention-based approach to detect the most interesting objects in an image. For example, saliency detection can be used as a pre-processing step to focus on specific regions of the image and afterwards process those regions. This often reduces the number of computations.

One requirement, for vision applications to be of use to a person, is that the application should be able to run in real-time. For this a lot of processing power is needed and therefore a hardware solution becomes an interesting option. The Field-Programmable Gate Array (FPGA) is suitable for this. Although it has a low clock frequency, the FPGA can still outperform other systems if parallelism is exploited.

But making an FPGA design in a hardware description language (HDL) is a time consuming job, because errors can be easily made and are often difficult to debug. Recently, a number of high-level synthesis (HLS) tools have become available and can be an alternative option to use. These tools take as input an application which is written in a high level language, such as C, and creates automatically the FPGA design with some directions from the designer.

Therefore, a case study has been done for a saliency detection algorithm using an HLS tool. The FPGA implementation is made using accelerators and each accelerator is designed to speed-up a specific part of the application. The advantage of using accelerators is the versatility in designing a system where the designer can choose which accelerators to use depending on the requirements. Using accelerators also creates a pipelined design, where each accelerator can start as soon it is finished and new data is available. This results in a much higher throughput.

Further, we analyze the possibilities of using the HLS tool with a skeleton approach. The idea of a skeleton is that certain functions can be abstracted, which results in more architecture independency. The functions are divided into several classes, such as a pixel-to-pixel class or neighborhood-to-pixel class. For these classes, an optimal skeleton can be made. In other words, given a specific class and the function it needs to perform, efficient RTL code can be generated for FPGAs using this skeleton structure. In this research, we focused on the pixel to pixel class and the neighborhood to pixel class.

Contributions: Following are the contributions of this paper:

- Mixed design approach of using a HLS tool and manual work for accelerators
- Full implementation of saliency detection on FPGA using floating point
- A comparison between a manual design and a design created by an HLS tool of an accelerator
- Analysis of AutoESL for skeletons

The structure of the paper is as follows. We begin in section II with the related work and explain in section III the algorithm used for saliency detection. Thereafter, section IV describes the implementation of the accelerators on FPGA for the algorithm. The analysis of the HLS tool AutoESL for floating point accelerators and skeletons is in section V and the results of the application implemented on FPGA are in section VI. We finish with the conclusion and recommendations for future research in section VII.

II. Related Work

A lot of research has been done on saliency detection [1] [2] [3] [4]. Furthermore, saliency detection is mostly used as a pre-processing algorithm to reduce the number of computations. For example, to scan the whole image for features can take quite some time, but only detecting features at certain regions can improve this as shown by Kim *et al.* [5]. However, they use the saliency detection algorithm from Itti *et al.*, whereas we use the algorithm from Hou *et al.* which has a better accuracy. Further, we use the idea of accelerators and skeletons to implement the saliency detection algorithm on FPGA as opposed to a chip design with a configurable heterogeneous multicore architecture.

For feature detection there is Feature-Accelerated Segment Testing (FAST) [6] [7], Scale-Invariant Feature Transform (SIFT) [8], and Speeded Up and Robust Feature (SURF) [9]. These feature detection algorithms are often used in vision applications as a front end. For example FAST is used in the vision application Parallel Tracking And Mapping (PTAM) [10]. Moreover, an FPGA implemenation of FAST and SURF already exists [11].

Further, HLS tools have been used in past research. Catapult C has been used to design an FPGA implementation fast [12] [13]. The tool works quite well, but does not support floating point which makes AutoESL a more suitable tool. Also, independent research has been done to evaluate HLS tools [14] [15]. From their research they concluded that AutoESL performed quite well.

Furthermore, application designers often need to have a understanding of the hardware to fully utilize its computing power. But this leads to more design complexity and often takes a lot of time if the designer is not familiar with the hardware architecture. Therefore, Wouter Caarls proposed to use algorithmic skeletons to achieve architecture independence for the embedded image processing domain [16].

Functions can be categorized in different classes, such as binarization which belongs to the pixel to pixel class or a 2D image filter which is a neighborhood to pixel class. Each class has its own algorithmic skeleton and are written in a algorithm-specific language. These skeletons are then transformed to a hardware target-specific language. This idea has been used for GPUs [17]. In this paper, we analyze whether the same idea of using skeletons is applicable to FPGAs.

III. Saliency detection algorithm

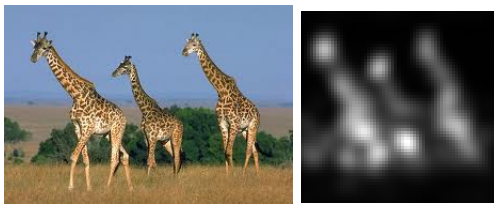
Saliency detection is based on how a human views an image. When a persoon sees an image, he tends to focus on the parts which stands out the most in an image. In Fig. 1, some examples of saliency are shown. The giraffes in the upper left image and the castle in the water in the bottom left image draw quickly attention, whereas the ground, sky and water are less interesting. Therefore, the giraffes and castle stand out the most and are recognized as salient objects in the images.

The algorithm chosen is the saliency detection algorithm from Hou *et al.* [4]. Their algorithm has good results in terms of accuracy and performance. In their approach they believe there is a connection between saliency and frequency. By taking the logarithm of the frequency spectrum, the log spectrum is obtained. Afterwards, by averaging this log spectrum and subtracting it from the original log spectrum, you get the spectral residual spectrum. They believe that the peaks in this spectrum have a correlation to the salient objects in the image and the results are quite good as shown in Fig. 1b and Fig. 1d.

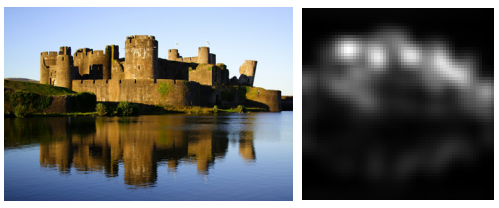
This algorithm is available as Matlab code, but for further conveniences the algorithm is ported to C and a small alteration has been applied to speed it up. In their algorithm they first convert the RGB colors to gray and then resize the image. By swapping these two functions, there is a reduction in the number of operations that need to be performed as shown in Table I. The resize algorithm uses a window of 4 by 4 and is executed as two 1D kernels. Therefore, the operations per pixel is multiplied by 5. The steps performed in the algorithm are then as follows:

1) Resize image to 64x64
2) Convert RGB colors to gray
3) Fourier transform image
4) Take logarithm of the absolute value of the Fourier transformed image
5) Use an 3x3 average filter on the log spectrum
6) Subtract the averaged log spectrum from the original log spectrum
7) Calculate back to complex numbers
8) Inverse Fourier transform
9) Use an Gaussian filter to smooth the image

In Table II each function has been classified. Most functions belong to the pixel to pixel class and some to the neighborhood to pixel class. Further, there is no good class yet for the 2D Fourier transform, but currently we classified it as a tile to tile class. It should be noted that the tile to tile class is too generic as all kernels fit into this class.



(a) Original image (b) Saliency image



(c) Original image (d) Saliency image

Fig. 1. Examples of saliency detection

IV. FPGA implementation

This application has been implemented on the Virtex-6 FPGA from Xilinx and the frequency used is 100 MHZ.

TABLE I
NUMBER OF OPERATIONS FOR RESIZING AND RGB TO GRAY CONVERSION

Function	Ops/pixel	Input size (First RGB to Gray conversion)	Number of operations	Input size (First resize)	Number of operations
RGB to gray conversion	5	300x400	600,000	64x64	20,480
Resize image	$5 \cdot 13$	64x64	266,240	3·64x64	798,720
Total			866,240		819,200

TABLE II
CLASSIFICATION KERNELS

Kernel	Class
Resize image	Neighborhood to pixel
RGB to gray conversion	Pixel to pixel
2D Fourier transform	Tile to tile
Logarithm	Pixel to pixel
Average filter	Neighborhood to pixel
Subtraction	Pixel to pixel
Complex number calculation	Pixel to pixel
2D Inverse Fourier transform	Tile to tile
Gaussian Filter	Neighborhood to pixel

Further, the accelerators are designed to be as fast as possible using not too much area of the FPGA. The FPGA implementation of this application uses a MicroBlaze as a base. The task given to the MicroBlaze is, to fetch the image from the compact flash card, to send the data to the accelerators, and to retrieve the processed image.

Furthermore, the MicroBlaze and the accelerators communicate through Fast Simplex Links (FSL), which are 32 bits point-to-point links with a FIFO buffer. Furthermore, all accelerators use floating point as input and output as it is easier to to fully utilize the bandwidth of the FSLs. Fixed point models can be made, but the exploration is extensive as every kernel uses floating point operations. Moreover, the impact on the end result is difficult to predict because in the end the final result is converted back to integers. Therefore, fixed point models are out of the scope of this research.

A. Topology of FPGA implementation

Fig. 2 shows the topology of the implementation of the application on FPGA. The accelerators that are orange are designed using the high level synthesis tool AutoESL and the green accelerators are manually designed. AutoESL does not support al functions, such as logarithm and the exponential function. Therefore, these are manually designed. Further, with AutoESL the Fourier transformation could not be made as efficient as a manual design.

Some of the steps in the algorithm are merged together to make use of the local bandwidth or data. This is the case for resizing the image and RGB to gray conversion, computing the absolute values and arguments of the complex numbers, average filtering and subtraction.

After the Fourier transform the data is split up into the absolute value of the complex number and the input of the inverse tangent accelerator to calculate the argument. These are later combined again into the sine and cosine accelerator to compute the complex numbers again needed for the invserse Fourier transform.

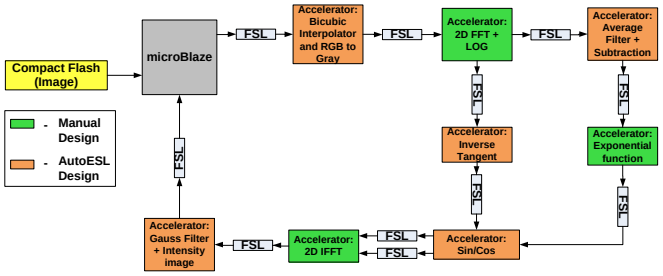


Fig. 2. Topology: Saliency detection FPGA

B. Resize image and RGB to gray conversion accelerator

Depending on the scale of the objects in the image, the salient objects can be different. For example, if there is a house in the image, the house can be a salient object. But if you look at it more closely, the doors and windows could also be seen as salient objects. Therefore, the image is resized to 64 by 64 pixels to omit these smaller objects.

The algorithm chosen to resize the image is bicubic interpolation, because it preserves fine detail in images better than bilinear interpolation. However, this comes at the cost of more computing power. Bicubic interpolation is implemented as two 1D operations that operates on a 4 by 4 neighborhood. Eqn. 1 shows the formula for 1 dimension where p_1 till p_4 are the input pixels and x is the coordinate of the interpolated pixel. For 2D this needs to be done for each row and then again with the 4 outputs to calculate the interpolated pixel.

$$\begin{aligned} t_1 &= (p_3 - p_4) - (p_1 - p_2) \\ t_2 &= (p_1 - p_2) - t_1 \\ t_3 &= p_3 - p_1 \\ P_{interpolate}[x] &= t_1 \cdot x^3 + t_2 \cdot x^2 + t_3 \cdot x + p_2 \end{aligned} \quad (1)$$

RGB to gray conversion is a simple operation where each color contributes a certain percentage to the gray pixel value. This formula is given in Eqn. 2.

$$P_{gray}[x,y] = 0.3 \cdot P_r[x,y] + 0.59 \cdot P_g[x,y] + 0.11 \cdot P_b[x,y] \quad (2)$$

In Fig. 3 the design is shown. We assume the image is stored on a compact flash card on the FPGA board, but this could be replaced by a camera link which directly feeds the input images to the accelerators. The image on the compact flash card is read out per 4 bytes by the MicroBlaze to fully use the bandwidth of the FSL links. Hereafter, the MicroBlaze sends the image to the image resizer and RGB to gray accelerator, which has 3 block RAMs to store the image. Each color is stored into a separate BRAM.

In AutoESL we chose this design to be pipelined to have a fast accelerator which does not use too much area. This gives a pipeline depth of 108 stages with an initiation interval of 8 cycles. The initiation interval is the amount of cycles it takes before it starts to calculate a new output. The BRAM is the bottleneck here as it only has 2 ports and it needs to load 16 pixels. Therefore, this results in an initiation interval of 8 cycles. Further, the large number of stages is caused by many dependencies and the use of floating point units. A multiplication or an add already costs 4 stages. Interested readers can refer to Appendix A1 for more details on this.

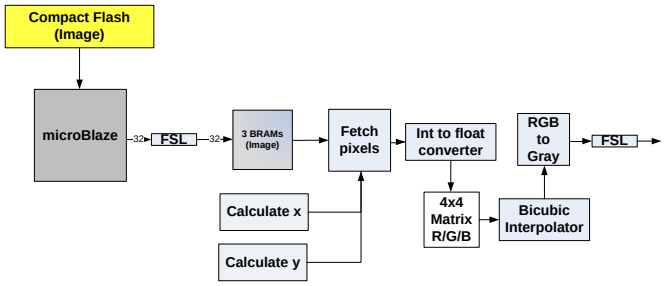


Fig. 3. Bicubic interpolator and RGB to Gray accelerator

C. 2D Fourier transform and logarithm accelerator

Fourier transform is performed using Fast Fourier Transform (FFT), which is efficient to implement on FPGAs. For this manually designed accelerator we used Xilinx Core generator to generate an FFT core and Radix-4 is chosen as architecture, because it is more efficient to use on a 64 points input.

The design is shown in Fig. 4. In this design, the accelerator assumes the data will be given column wise to save memory at the end of the accelerator. Each column will first be transformed and stored into BRAM. Thereafter, each row will be transformed to complete the 2D Fourier transform.

To save area and make use of the local bandwidth, there is some post processing on the complex numbers which is pipelined. In the top branch in Fig. 4, the absolute value of the complex number is calculated and passed onto the logarithm functional unit. In the bottom branch the accelerator already calculates the input of the inverse tangent accelerator, which is needed to compute the argument shown in Eqn. 3.

$$\phi = arg(P_{FFT}[x,y]) = arg(a+ib)$$
$$arg(a+ib) = \begin{cases} 2\arctan\frac{b}{\sqrt{a^2+b^2}+a} & \text{if } a>0 \text{ or } b\neq 0 \\ \pi & \text{if } a<0 \text{ and } b=0 \\ \text{undefined} & \text{if } a=0 \text{ and } b=0 \end{cases} \quad (3)$$

Because AutoESL does not support the logarithm operator, we used the FloPoCo generator [18] to implement this function. Further, this accelerator does not support synchronization signals which makes it difficult to use the pipeline. Therefore, it is merged with the 2D Fourier Transform accelerator which guarantees that there is an output every cycle when the FFT core is ready with the transform. When the output of the logarithm functional unit changes, 64 outputs are taken from the logarithm functional unit.

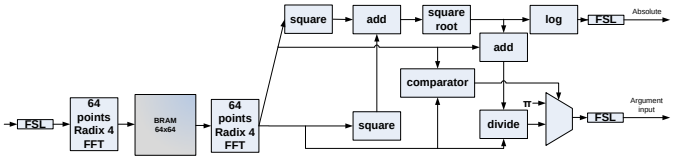


Fig. 4. 2D FFT accelerator

D. Average filter and subtraction accelerator

In this accelerator we take the output of the 2D FFT and logarithm accelerator and use a 3x3 average filter on it. Because the filter is separable, the average filter is implemented using two 1-dimensional convolution units as shown in Eqn. 4. Further, if this 3x3 window is outside the image range, the nearest pixel is taken as shown in Eqn. 5.

$$\begin{aligned} P_{avg}[x,y] &= \sum_{i=-1}^{1}\sum_{j=-1}^{1} P_{log}[x-i,y-j]\cdot h[i,j] \\ &= \frac{1}{9}\sum_{j=-1}^{1}\sum_{i=-1}^{1} P_{log}[x-i,y-j] \end{aligned} \quad (4)$$

$$P_{avg\_1D}[x] = \begin{cases} 2P[x]+P[x+1] & \text{if } x=0 \\ 2P[x]+P[x-1] & \text{if } x=\text{WIDTH}-1 \\ P[x-1]+P[x]+ & \\ P[x+1] & \text{others} \end{cases} \quad (5)$$

The design is shown in Fig. 5. Two multiply-adds are drawn to illustrate the 3 options, but actually there is only one in the implementation. Depending on the x-coordinate, the correct pixels are loaded from the BRAM.

The accelerator works as follows. The input of the accelerator is first put into BRAM. Thereafter, the rows are added to each other and will continue till the 2nd BRAM is filled. This is the initialization part. After the initialization of this BRAM, the columns in this BRAM can be added to each other and the result is divided by 9 to complete the 2 dimensional convolution. Finally, the averaged results is subtracted from the original input. While it is adding the columns, a new row is already being calculated simultaneously and put into the 2nd BRAM.

For this initialization part, the initiation interval of the pipeline is 2 cycles, because it needs to load 3 pixels but the BRAM only has 2 ports. Further the pipeline depth for this is 13 stages.

The main part of the accelerator is a bit more complex and has an initiation interval of 5 cycles and the pipeline depth is 25 stages. Interested readers can refer to Appendix A1 for more details.

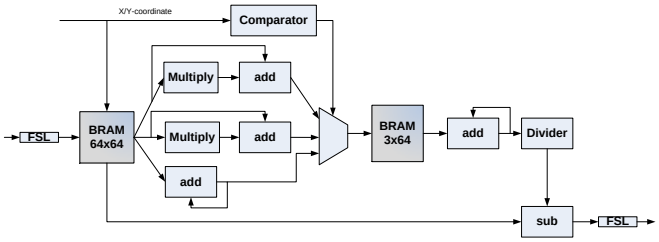


Fig. 5. 3x3 Average filter and subtraction accelerator

E. Calculate complex number accelerator

The equation to calculate the complex number is given in Eqn. 6, where P_{res} is the magnitude of the spectral residual spectrum.

$$P_{complex}[x,y] = \exp^{P_{res}[x,y]}(\cos\phi + i\sin\phi) \tag{6}$$

The first part is to calculate the magnitude of the complex number. The logarithmic scaled values of the spectral residual spectrum are scaled back by using the exponential function. AutoESL does not support this, therefore we use the floating point core from the FloPoCo generator [19].

The second part of calculating the complex number is to obtain the argument ϕ of the resized image. As shown in Eqn. 3, the inverse tangent operation is needed. Trigonometric functions are hard to implement on an FPGA, therefore an approximation is used. Two approximations have been considered to perform the inverse tangent. The first one is to use a Taylor series which is shown in Eqn. 7. The second one is to use the approximation shown in Eqn. 8 [20].

$$\arctan(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} \qquad \text{for } -1 \leq x \leq 1 \tag{7}$$

$$\arctan(x) = \frac{x}{1 + 0.28125x^2} \qquad \text{for } -1 \leq x \leq 1 \tag{8}$$

In Fig. 6, the error percentage is plotted for the inverse tangent approximations. The 15th order Taylor series approximation has a low error percentage in the region -0.8 to 0.8, but beyond that the error increases exponentially. The other inverse tangent approximation has overall a low error percentage with maximum 1% error. Therefore, this second approximation has been chosen to use for the inverse tangent. Eqn. 9 shows how this approximation can be used for other regions.

For the inverse tangent accelerator this results in an initiation interval of 1 cycle and a pipeline depth of 46 stages. Interested readers can refer to Appendix A1.

$$\arctan(x) = \begin{cases} \arctan(x) & \text{for } -1 \leq x \leq 1 \\ -\frac{\pi}{2} - \arctan\frac{1}{x} & \text{for } x < -1 \\ \frac{\pi}{2} - \arctan\frac{1}{x} & \text{for } x > -1 \end{cases} \tag{9}$$

With the argument and magnitude, the complex number can be calculated using the sine and cosine. For these functions a Taylor series approximation is used (Eqn. 10 and Eqn. 11). In Fig. 7, the error percentage is plotted for a 13th order sine approximation and a 12th order cosine approximation. The

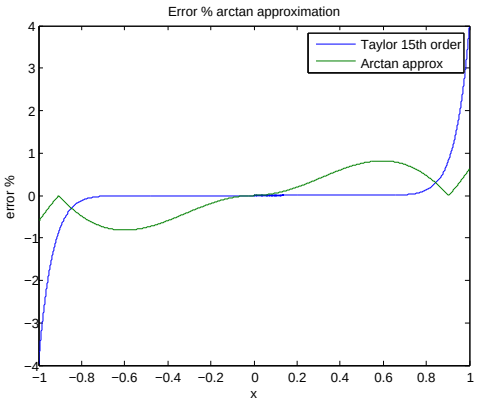


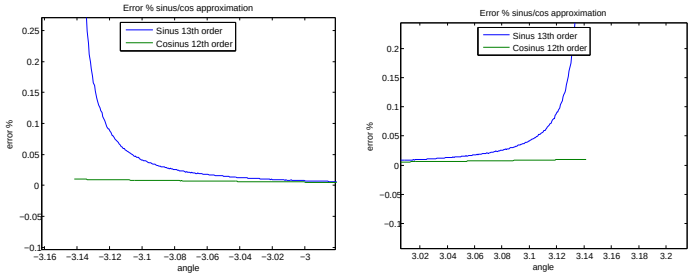
Fig. 6. Error % inverse tangent approximation

errors are quite small but increase when the angle approaches $-\pi$ and π. These two approximations have been chosen, because they are easy to implement. The even terms can be used for the cosine approximation and the odd terms for the sine approximation.

This accelerator has an initiation interval of 1 cycle and a pipeline depth of 66 stages. Interested readers can refer to Appendix A1 for more details.

$$\sin(\phi) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} \phi^{2n+1} \tag{10}$$

$$\cos(\phi) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} \phi^{2n} \tag{11}$$



(a) Error % left (b) Error % right

Fig. 7. Error % sin/cos approximation

F. 2D Inverse Fourier transform accelerator

With the complex numbers obtained from the previous step, the inverse 2D Fourier Transform can be executed. The same design as the 2D Fourier transform is used, with some modifications to the post-processing as only the absolute value is needed as output of this accelerator. The design is shown in Fig. 8. This accelerator has 2 FSLs, one is for the real part of the complex numbers and the other one for the imaginary part. The input data is first transformed for the rows and then columns. It should be noted that, the output of this accelerator is now column wise, because the second Fourier transform is done on the columns of the data. Therefore an extra transpose step is needed, which is performed as part of the Gaussian filter accelerator.

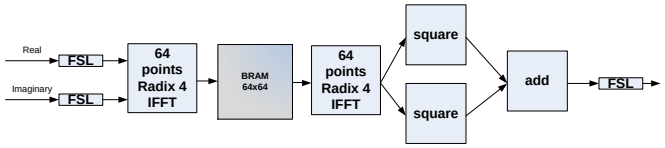


Fig. 8. 2D IFFT accelerator

G. Gaussian filter accelerator

With the inverse Fourier transformed image, the image now needs to be smoothen out to see the salient objects better. For this a 2D Gaussian filter is used with a window of 10 by 10. Just as the average filter, the gaussian filter is also separable into two 1D filters. Eqn. 12 shows the equation for the convolution filter.

$$\begin{aligned} P_{gauss}[x,y] &= \sum_{i=-4}^{5}\sum_{j=-4}^{5} P_{IFFT}[x-i,y-j]\cdot h[i,j] \\ &= \sum_{j=-4}^{5} h_1[j] \sum_{i=-4}^{5} P_{IFFT}[x-i,y-j]\cdot h_1[i] \end{aligned} \tag{12}$$

In Fig. 9 its design is shown. The accelerator has a similar design as the average filter, but it handles the pixels around the border differently. Non-existing pixels are 0 if the window is outside the range of the image. Further, we choose the 2nd BRAM to have space for 16 rows to avoid the use of a modulo operation.

The initialization part needs an initiation interval of 5 cycles, as there are only 2 ports for the BRAM, and the pipeline depth is 56 stages. The main part of the filtering has an initiation interval of 7 cycles and a pipeline depth of 96 stages. Interested readers can refer to Appendix A1 for more details on this.

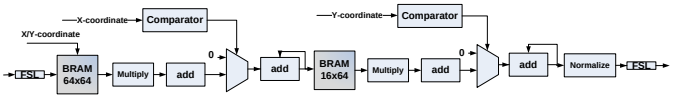


Fig. 9. 10x10 Gaussian filter and normalization

After the Gaussian filter, the image is normalized to the range 0 and 1. Eqn. 13 shows how this is done.

The initiation interval of finding the minimum and maximum is 2 cycles and the pipeline depth is 5 cycles. The normalization has an initiation interval of 1 cycle and the pipeline depth is 10 stages.

$$\begin{aligned} Max &= \max(P_{gauss}) \\ Min &= \min(P_{gauss}) \\ \delta &= \frac{1}{Max - Min} \\ b &= -min\cdot\delta \\ P_{saliency}[x,y] &= P_{gauss}[x,y]\cdot\delta + b \end{aligned} \tag{13}$$

V. Analysis of AutoESL

AutoESL takes C/C++ or SystemC as an input and generates the RTL code for an FPGA. FPGAs from Xilinx are all supported and it can generate floating point units. Further, optimizations can be applied with directives or with pragmas in the code. The tool is also easy to use and therefore AutoESL looks to be a viable candidate for the skeleton approach.

In this section, we are analyzing the HLS tool AutoESL on how well it performs and the suitability for skeletons. First, a comparison is done between a manual implementation of the RGB to gray accelerator and an implementation made using AutoESL to analyze the tool. Further, we analyze the design space exploration capabilities of AutoESL. We investigate if it possible to have a more configurable skeleton where different design goals can be chosen, such as area or throughput. Thereafter, we apply the skeleton idea to the pixel to pixel and neighborhood to pixel class.

A. RGB to gray analysis and comparison

In section IV-B Eqn. 2, the formula for converting an RGB pixel to gray is given. In the manual implementation, we assume a design with an operating frequency of 100 MHz and 1 FSL link. Therefore, it takes 3 cycles to receive 1 floating point RGB pixel. Further, a pipelined design is chosen to get the cycle diagram shown in Fig. 10.

The Xilinx core generator is used to generate the floating point units and the units are also pipelined. Further, some extra control logic is used to shorten the latency in two other situations than the one shown in the figure. One situation is when the multiplication of the blue value is finished at the same time as the add of red and green. In that case, the register for blue is bypassed. The other situation is when the add of red and green is finished, but the multiplication of blue is not. In that case, we need to wait for the multiplication to finish and thereafter the register can also be bypassed.

The same pipelined design can be achieved in AutoESL and AutoESL also uses the core generator to generate the floating point units. Fig. 11a shows a screenshot of the schedule that AutoESL produces. But by looking at the VHDL code that is generated, the actual cycle diagram can be drawn as shown in Fig. 11b.

Apparently, AutoESL needs a cycle to put the result of a floating point unit into a register and every floating point operation has its own register. Further, there is an extra register (Register4) to ensure the correct data is passed on. The blue value is added in cycle 10, but register 3 already gets overwritten in cycle 9. Hence, the need for the extra register.

Furthermore, AutoESL adds a 1 bit control signal for every register to indicate if the data in the register is valid or not. The generated VHDL has some structure, but it quite difficult to discover the schedule. It produces a large finite state machine with a process for each register, floating point unit and control signal. In these processes, the signals are assigned the correct values depending on the state and control signals. Therefore, modifying the generated VHDL code will be a difficult task.

The overhead caused by using an HLS tool seems quite small and the design time is much faster using AutoESL.

Further, the designer does not need to put effort anymore in designing the control unit.

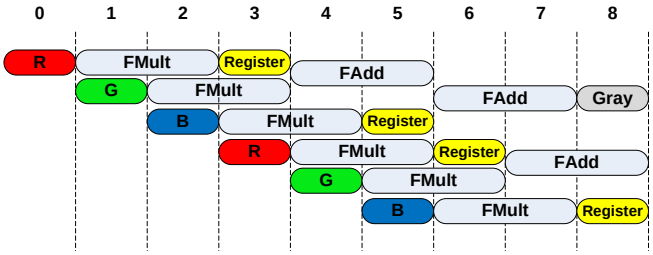
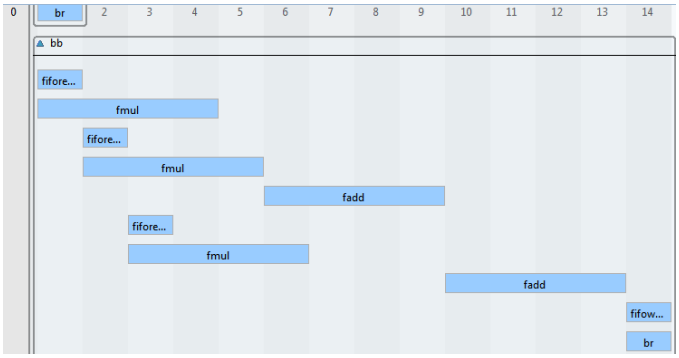
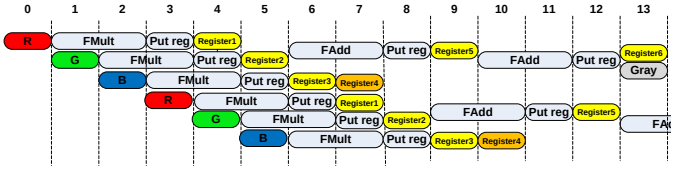


Fig. 10. Manual: Cycle diagram RGB to gray accelerator



(a) Screenshot schedule AutoESL



(b) Actual cycle diagram RGB to gray accelerator

Fig. 11. AutoESL schedule RGB to gray accelerator

Table III shows is a comparison between the manual and AutoESL version of the RGB to gray accelerator. It takes about 6 hours to implement the accelerator and about half an hour using AutoESL. The most time consuming job is mostly to debug the logical errors one can make while writing the VHDL code, because some situations might have been overlooked. Further, AutoESL has a longer pipeline depth due to the use of registers. The number of look-up tables are comparable, but some extra control logic is used in the manual design to control the adding of blue in the two other situations to shorten the latency. The difference in the number of flip flops can be explained by the number of registers. The difference is about $\frac{473-358}{32} \approx 3.6$ registers. The manual design uses 3 registers: 1 to store red, 1 store blue and 1 to store the addition of red an green. And the accelerator designed by AutoESL uses 6 registers as explained earlier. The other flip flops are probably from control signals. The number of DSPs are the same, because the floating point units are configured in the same manner.

In the last row, the execution time is shown for converting 10 RGB pixels to gray values. There is a small overhead due to the MicroBlaze, but the overhead is the same in both designs. The difference in latency between both designs is 7 cycles. The accelerator made by AutoESL needs an extra cycle to enter the loop control and 1 cycle to exit it. The other 5 cycles are due to the difference in pipeline depth.

TABLE III
COMPARISON MANUAL AND AUTOESL RGB TO GRAY ACCELERATOR

	Manual	**AutoESL**
Design time	6 hours	30 minutes
Pipeline depth	9	14
LUTs	553	590
Flip Flops	358	473
DSPs	5	5
Execution time (10 pixels)	134 cycles	141 cycles

B. Design space exploration using AutoESL

Different designs can be evaluated and often a trade-off is required depending on the design goal. This is also the case for skeletons. One can choose to have the fastest design at the cost of more area or a slower design which uses less area.

In AutoESL it is possible to do a quick design space exploration. For this, some arbitrary functions are taken with 1, 3, 5 and 7 operations as shown in Eqn. 14 till 18. In these equations, W are some constant weights, which are multiplied with the inputs and then added to each other. Eqn. 18 is the same as Eqn. 17, but written differently. In the design we pipeline these functions to accelerate them.

$$y = W_1 \cdot a \tag{14}$$
$$y = W_1 \cdot a + W_2 \cdot b \tag{15}$$
$$y = W_1 \cdot a + W_2 \cdot b + W_3 \cdot c \tag{16}$$
$$y = W_1 \cdot a + W_2 \cdot b + W_3 \cdot c + W_4 \cdot d \tag{17}$$

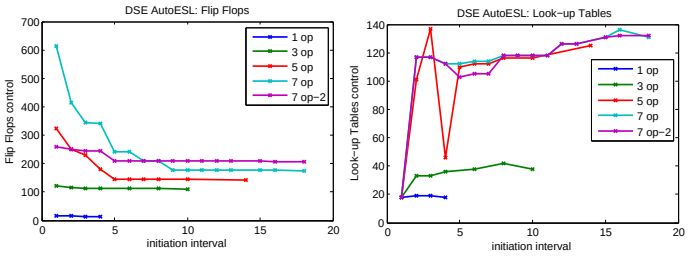
$$\begin{aligned} t_1 &= W_1 \cdot a + W_2 \cdot b \\ t_2 &= W_3 \cdot c + W_4 \cdot d \\ y &= t_1 + t_2 \end{aligned} \tag{18}$$

In Fig. 12, we plotted the initiation interval against the amount of flip flops and look-up tables used for the control logic and the amount DSPs used. As mentioned earlier, the initiation interval is the amount of cycles it takes before it starts to calculate a new output. For example, the initiation interval in the RGB to gray accelerator in section V-A is 3 cycles, because it takes 3 cycles to read the red, green and blue value. But if more bandwidth is available, by using 3 FSL links, an initiation interval of 1 can be achieved. However, this comes at the cost of more flip flops and DSPs as shown in the figures. But the amount of look-up tables for control is decreased. The reason for this is that some floating point units can no longer be reused and the floating point units need to be duplicated. The look-up tables that were used to put the correct data into the floating point units are now no longer needed anymore.

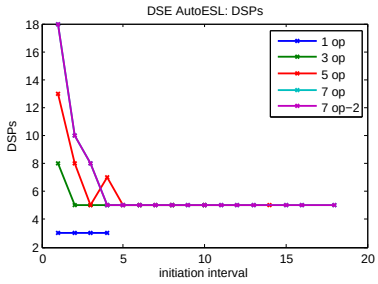
Further, as shown in the graph the number of flip flops and DSPs decreases when the initiation interval is increased, because more reusage of the floating point units are possible and fewer registers are needed. But increasing the initiation interval leads to a higher execution time. Therefore, a trade-off needs to be made between area usage and speed.

In Fig. 12b, we notice a dip for the red line (5 op). This can be explained by the cycle diagram in Fig. 13. In cycle 10 there is an overlap with the two additions. Therefore, two floating point adders are generated, which can also be seen in the increase of DSPs used in Fig. 12c. Therefore, look-up tables to control the input of the adders are no longer needed. In this case, a lower initiation interval does not result in less area.

Another thing to notice, is the difference between the purple and light blue line. Although it is the same function, but they are written differently. The reason is that AutoESL likes to wait till the addition finishes until the next addition can be performed. By writing it like Eqn. 18, we guide AutoESL in not doing it. Further, the number of flip flops seems constant over all the initiation intervals in this case. This is because fewer pipeline registers need to be inserted, but AutoESL still generates a register for every operation as explained in section V-A.



(a) DSE Flip Flops (b) DSE Look-up Tables



(c) DSE DSPs

Fig. 12. Design space exploration

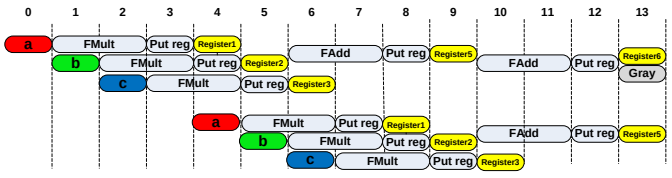


Fig. 13. Cycle diagram 5 operations, II=4

C. Using Skeletons with AutoESL

We consider 4 trajectories to design an FPGA implementation, which are shown in Fig. 14. Some of the trajectories use a skeleton. By using skeletons, design time is shortened and programmers do not need to have a full understanding of the architecture to make an efficient design.

The first trajectory is to manually implement a function for FPGA in HDL. The second trajectory is to manually convert the function to HDL code and insert that into an HDL skeleton. This can then be used to generate the FPGA implementation. The third trajectory is to manually re-code the function to an intermediate representation, which we call C'. C' contains optimizations to create an efficient FPGA design using a high level synthesis tool. The fourth trajectory uses skeletons. These skeletons are optimized for a certain class and only the functionality of the function and some parameters need to be inserted into the skeleton to generate C'. Thereafter, C' is used in a high level synthesis tool to generate the FPGA design.

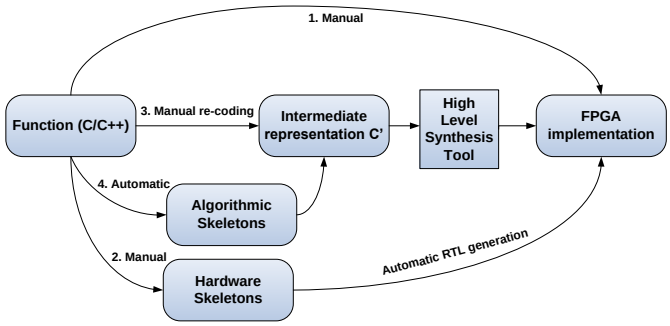


Fig. 14. FPGA design trajectories

The first approach is time consuming and many simple errors can be made. The second option is still a lot of work. The function and the control unit still need to be manually written, which is actually the most work when manually writing the HDL code. For example in the RGB to gray accelerator, the scheduling is fairly easy, but the control unit needs to consider all situations for correct behavior. Further, the HDL code needs to be rewritten if more FSLs are used.

In this subsection, we discuss whether the third and fourth approach are possible to do with AutoESL. We mainly focus on the pixel to pixel and neighborhood to pixel class and floating point, because most functions in the saliency detection algorithm use floating point and belong to one of those two classes.

From the previous sections, we can already conclude that the coding style greatly affects the generated schedule and different architectures can be created. Further, there is an overhead in area usage. For example, AutoESL generates a 32 bits register for every floating point operation.

However, there is a lot of flexibility in AutoESL. For the pixel to pixel class, the functions can be easily replaced with another function from the same class. In Code 1, we show an example of a skeleton instantiation for the RGB to gray accelerator. From the function *p2p_skeleton()*, it can be seen that there is 1 incoming (*input*) and 1 outgoing (*output*) FSL.

If the designer chooses to use more FSLs, more arguments should be given to the function. The input data first needs to be copied to a variable, because the data is interleaved into the FSL link. Thereafter, the function call is made to convert the RGB pixel to a gray value. Further, the pipeline directive is applied to *LOOP* and the FIFO and the FSL directive are applied to *input* and *output*.

The *rgb2gray()* function can be easily replaced with the *max()* function shown in Code 2. We compare red and green first and the largest is stored into a temporary register. Thereafter, we compare that result with blue and take again the largest. The cycle diagram is shown in Fig. 15. It should be noted that, the function call and data copying can also be replaced directly with the function itself in the main code, but in this way the skeleton idea is easier to understand.

```
1  void p2p_skeleton(float input[3*N], float
       output[N])
2  {
3    int i, counter = 0;
4    float input1, input2, input3;
5
6    LOOP:
7    for(i = 0; i < N; i++) {
8      input1 = input[counter];
9      input2 = input[counter+1];
10     input3 = input[counter+2];
11     output[i] = rgb2gray(input1, input2,
           input3);
12     counter += 3;
13   }
14 }
15
16 float rgb2gray(float r, float g, float b)
17 {
18   return 0.3*r + 0.59*g + 0.11*b;
19 }
```

Code Example 1. RGB to gray skeleton instantiation

```
1  float max(float r, float g, float b)
2  {
3    float temp1;
4    if(r > g) temp1 = r;
5    else temp1 = g;
6
7    if(b > temp1) return b;
8    else return temp1;
9  }
```

Code Example 2. maximum operator code

Other functions are also possible. Instead of adding and multiplying in the RGB to gray case, subtracting and dividing is also possible. This leads to a similar schedule, but division takes more cycles which results in a longer pipeline depth and might also result in using more registers. However, not all mathematical operations are available. Taking powers, exponential, logarithm and trigonometric functions are not possible.

Automatic generation of C' is possible using skeletons for the pixel to pixel class, because it has a fixed structure. But sometimes the function is not scheduled optimally if there are more than 2 dependencies as shown in section V-B by Eqn. 17 and Eqn. 18. A solution would be to also transform the code and split up an equation into multiple equations if there are more than 2 dependencies. But this mostly affects only the number of flip flops used and not throughput.

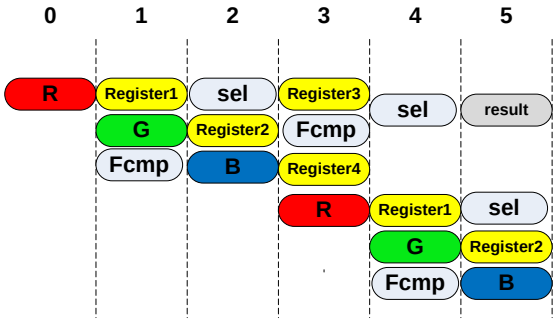
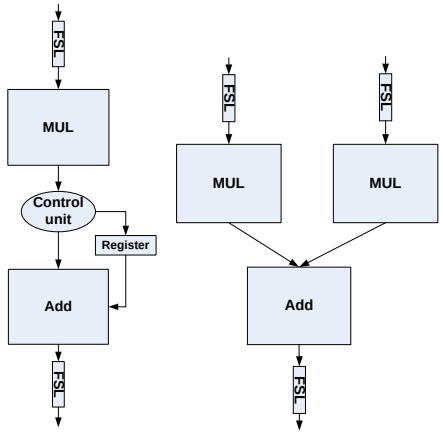


Fig. 15. Cycle diagram $\max(R, G, B)$

Further, besides image size, the class and function, the designer should also give the number of FSL links ingoing and outgoing of the accelerator as a parameter to instantiate the skeleton. Depending on the bandwidth available, the schedule and architecture can be produced differently as shown in the previous subsection. For example, Fig. 16 shows 2 different architectures for the same function given in Eqn. 19, with different incoming bandwidth. c_1 and c_2 are constants. Each architecture has its own advantage and disadvantage. The architecture in Fig. 16a only needs 1 floating point multiplier and 1 floating point adder, but is twice as slow as the architecture in Fig. 16b. However, the second architecture uses one extra floating point multiplier.

$$y(x, z) = c_1 \cdot x + c_2 \cdot z \tag{19}$$



(a) 1 FSL link (b) 2 FSL links

Fig. 16. 2 different architectures for the same function

For the neighborhood to pixel class, AutoESL can also be used. The same structure of performing two 1D convolutions for 2D filters has been used for the average filter and Gaussian filter in the saliency detection algorithm. But instead of convolution, a comparison function that compares the center pixel with its neighbors could also be used. However, this skeleton is only suitable for a specific subclass, where a 2D function can be split up into two 1D functions. Further, some

optimizations can still be made. The BRAM is currently the bottleneck as it only has 2 ports. The Gaussian filter has a window of 10 by 10. Therefore, ideally you want to have 10 ports by using multiple BRAMs or reshape the BRAM to load multiple pixels with 1 access. But this comes at the cost of using more area as more multipliers and adders are needed in parallel.

Further, some specific optimizations can be applied to the average filter. Currently, all the pixels are added to each other again when the filtered pixel is calculated. But the previous result can be reused by subtracting the first pixel of the previous window and add the new pixel when the window shifts by 1. This reduces the number of loads and operations. Therefore, multiple skeletons can exist for 1 class where each skeleton makes use of different optimization techniques that can be applied.

VI. Results

This section presents the results of implementing the saliency detection onto FPGA. We compare the execution times of the application on a CPU, MicroBlaze and the FPGA accelerators. The execution times on CPU are measured using a laptop with a Core i5 processor at 2.53 GHz and 4 GB of RAM. The MicroBlaze and the accelerators use a frequency of 100 MHz. Furthermore, the MicroBlaze fetches the image from DDR memory for the resize kernel. Therefore, we added cache to the MicroBlaze to improve the execution time of the resize kernel. Thereafter, the resized image is stored into BRAM.

The FPGA accelerators were measured using a hardware timer controlled with the MicroBlaze. To measure the execution times of the accelerators separately, we connect each accelerator to the MicroBlaze instead of linking the accelerators to each other. This results in a longer execution time per accelerator, because the data is first send back to the MicroBlaze and then passed on to the next accelerator. Furthermore, the data is send and received in a for loop, which means the loop overhead is also timed.

Table IV shows the execution times of the application on CPU, MicroBlaze and the accelerators. The CPU needs 5.57 ms to detect salient objects in an image. The MicroBlaze does that in 316.5 ms and the FPGA accelerators need 1.80 ms. On the CPU and MicroBlaze, resizing the image takes the most time, but on the FPGA the Gaussian filter is the slowest kernel. This is because, the normalization is also part of the Gaussian filter accelerator. But in total the accelerators are still 3 times faster than the CPU and about 175 times as fast as the MicroBlaze. Though, it should be noted that for resizing the image, the MicroBlaze fetches data from DDR memory and the accelerator fetches data from BRAM.

The exponential and inverse tangent function are also mentioned, but are not included in the total timing of the application for CPU and MicroBlaze. The inverse tangent is actually merged with the loop where the logarithm is calculated. The exponential function and calculating the complex number are merged with the average filter loop. In this way, the execution time is reduced because there is less loop overhead.

Further, small deviations in the output image have been detected when it is compared with the image of the original algorithm. These deviations are because of the approximations used for the trigonometric functions. On average there are 171 pixels that are off by 1 value from the original output. However, these small deviations are not visible and do not affect whether an object is considered salient or not.

TABLE IV
Comparison execution time saliency detection

Kernel	CPU	MicroBlaze	FPGA accelerator
Resize and RGB to gray conversion	2.53 ms	147.36 ms	0.25 ms
FFT and logarithm	0.95 ms	50.6 ms	0.55 ms
Average filter and subtraction	0.69 ms	40.6 ms	0.37 ms
Exponential function	0.19[1]ms	24.1[1]ms	0.45 ms
Inverse tangent	0.58[1]ms	15.7[1]ms	0.33 ms
Complex number calculation	-[2]	-[2]	0.41 ms
IFFT	0.38 ms	32.2 ms	0.6 ms
Gaussian Filter	1.02 ms	45.7 ms	0.78 ms
Total	5.57 ms	316.5 ms	1.80[3]ms

[1] Timed separately and not part of algorithm timing
[2] Part of average filter and subtraction
[3] Pipelined connection

Table V shows the area usage and the pipeline depth of the accelerators and the MicroBlaze. The resize and RGB to gray accelerator uses the most area, about 12 times as much look-up tables as the MicroBlaze. In total the accelerator uses 22.8 times more flip flops, 28.2 times more look-up tables, 60.2 times more DSPs and 8.2 times more BRAMS than the MicroBlaze. Of the logic available on the Virtex-6, the accelerators use about 16.4% of flip flops, 54.4% of look-up tables, 47% of DSPs and 33.7% of BRAMs available. But the speed-up gained in return is much more.

Further, the accelerators are pipelined, which means the throughput is determined by the slowest accelerator. Given the results in Table IV, the slowest accelerator is the Gaussian filter and therefore more than 1000 frames per second is achieved. This number is in reality higher, because overhead due to the MicroBlaze is also measured. We designed the accelerators to be as fast as possible to see what the limits are, but for normal use 30 frames per second is sufficient. Therefore, it might not be necessary to process the data pipelined in the accelerator, but sequential processing might also be sufficient to meet the requirements. Sequential data processing in the accelerator results also in less area usage.

The time spend to design the accelerators and optimize it are in Table VI. The time taken to generate the whole FPGA design and linking every accelerator are not included. The accelerators made using AutoESL are all designed faster than the manual designed accelerators. One of the reasons that the AutoESL designs are fast to make, is because the C code has already been written and functionality of the accelerator can be easily checked with simulation which reduces debug time

TABLE V
AREA USAGE SALIENCY DETECTION FPGA ACCELERATORS

Kernel	Flip Flops	Look-up Tables	DSPs	BRAMs	Pipeline depth
MicroBlaze	2173	2910	6	17	5
Resize and RGB to gray conversion	15998	36298	146	89	108
FFT and logarithm	6764	7629	47	20	-
Average filter and subtraction	1802	2468	7	5	13, 25
Exponential function	256	546	1	1	1
Inverse tangent	4672	7215	37	0	46
Complex number calculation	9716	15552	66	0	66
IFFT	5825	6206	42	16	-
Gaussian Filter	4478	6097	15	9	56, 96, 5, 10
Total	49511	82011	361	140	
Logic available	301440	150720	768	416	
Utilization %	16.4%	54.4%	47%	33.7%	

significantly.

TABLE VI
DESIGN TIME FPGA ACCELERATORS

Kernel	Design time
Resize and RGB to gray conversion	3 days
FFT and logarithm	13 days
Average filter and subtraction	1 day
Exponential function	0.5 day
Inverse tangent	1 day
Complex number calculation	1 day
IFFT	8 days
Gaussian Filter	2 days
Total	29.5 days

VII. CONCLUSIONS AND FUTURE WORK

We presented an implementation of saliency detection on FPGA. Using the accelerator idea, the design of the system becomes more flexible. One can choose to use only a few accelerators or all accelerators. Further, we used an HLS tool to design several accelerators. Using all accelerators a speed-up of 3 times has been achieved compared to CPU and the pipelined design can process more than 1000 frames per seconds, but this takes about half of the area on the FPGA.

Further, we analyzed the use of AutoESL for the skeletons. For at least the pixel to pixel class and floating point, it performs quite well with a small overhead. For the neighborhood to pixel class, we made a skeleton for 2D convolution kernels which are separable into two 1D kernels. However, the dual-ported BRAM is currently the bottleneck in this skeleton.

The design time is drastically decreased compared to a manual approach, but still some caution is required. Some parts of the code might need to be rewritten to produce an optimal architecture and schedule.

Future work. We only analyzed AutoESL for one case, where most functions used floating point and are from the pixel to pixel or neighborhood to pixel class. In the future this research needs to be expanded to more functions and classes and other data types to cover most used functions in vision applications.

Further, improvements to the accelerators can still be made. In some of the accelerators, we used division by a constant but these can be replaced by multipliers at the cost of using more DSPs. But dividers cost more area in terms of look-up tables and flip flops, and take more cycles to finish.

The Gaussian filter accelerator can also be improved. Currently, this accelerator takes up most of the time, because of the amount of operations and BRAM accesses it needs. Reshaping the BRAM or multiple BRAMs can be used to calculate everything in parallel, but this comes at the cost of using more area.

The resize and RGB to gray accelerator can also be improved in terms of area. Currently the equations for bicubic interpolation have not been rewritten to guide AutoESL in using a better schedule. Doing this will reduce the pipeline depth and decrease area.

Furthermore, this research can be extended with fixed point models to reduce the amount of area used. A lot of look-up tables and DSPs are used, therefore this design might not fit into smaller FPGAs. Also, given a lower throughput requirement, the accelerators themselves may not need all optimizations. The data in the accelerator can be even processed sequentially if it meets the throughput requirement and by processing data sequentially less area is used.

REFERENCES

[1] L. Itti, C. Koch, and E. Niebur, "A model of saliency-based visual attention for rapid scene analysis," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 20, no. 11, pp. 1254 –1259, nov 1998.

[2] Z. Liu, Y. Xue, H. Yan, and Z. Zhang, "Efficient saliency detection based on gaussian models," *Image Processing, IET*, vol. 5, no. 2, pp. 122 –131, march 2011.

[3] B. Gao, Z. Kou, and Z. Jing, "Stochastic context-aware saliency detection," in *Computer and Management (CAMAN), 2011 International Conference on*, may 2011, pp. 1 –5.

[4] X. Hou and L. Zhang, "Saliency detection: A spectral residual approach," in *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, june 2007, pp. 1 –8.

[5] K. Kim, S. Lee, J.-Y. Kim, M. Kim, and H.-J. Yoo, "A configurable heterogeneous multicore architecture with cellular neural network for real-time object recognition," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 19, no. 11, pp. 1612 –1622, nov. 2009.

[6] E. Rosten and T. Drummond, "Fusing points and lines for high performance tracking." in *IEEE International Conference on Computer Vision*, vol. 2, October 2005, pp. 1508–1511. [Online]. Available: http://mi.eng.cam.ac.uk/~er258/work/rosten_2005_tracking.pdf

[7] ——, "Machine learning for high-speed corner detection," in *European Conference on Computer Vision*, vol. 1, May 2006, pp. 430–443. [Online]. Available: http://mi.eng.cam.ac.uk/~er258/work/rosten_2006_machine.pdf

[8] D. Lowe, "Object recognition from local scale-invariant features," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2, 1999, pp. 1150 –1157 vol.2.

[9] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Comput. Vis. Image Underst.*, vol. 110, pp. 346–359, June 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1370312.1370556

[10] G. Klein and D. Murray, "Parallel tracking and mapping for small ar workspaces," in *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, nov. 2007, pp. 225 –234.

[11] M. Kraft, M. Fularz, and A. Kasiński, "System on chip coprocessors for high speed image feature detection and matching," in *Proceedings of the 13th international conference on Advanced concepts for intelligent vision systems*, ser. ACIVS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 599–610. [Online]. Available: http://dl.acm.org/citation.cfm?id=2034246.2034307

[12] Y. Guo and D. McCain, "Rapid prototyping and vlsi exploration for 3g/4g mimo wireless systems using integrated catapult-c methodology," in *Wireless Communications and Networking Conference, 2006. WCNC 2006. IEEE*, vol. 2, april 2006, pp. 958 –963.

[13] T. Damak, I. Werda, N. Masmoudi, and S. Bilavarn, "Fast prototyping h.264 deblocking filter using esl tools," in *Systems, Signals and Devices (SSD), 2011 8th International Multi-Conference on*, march 2011, pp. 1 –4.

[14] Berkeley Design Technology, "An independent evaluation of: High-level synthesis tools for xilinx fpgas," 2010.

[15] ——, "An independent evaluation of: The autoesl autopilot high-level synthesis tool," 2010.

[16] W. Caarls, "Automated design of application-specific smart camera architectures," *PhD thesis, Delft University of Technology*, 2008.

[17] C. Nugteren, H. Corporaal, and B. Mesman, "Skeleton-based automatic parallelization of image processing algorithms for gpus," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, july 2011, pp. 25 –32.

[18] J. Detrey and F. de Dinechin, "A parameterizable floating-point logarithm operator for fpgas," in *Signals, Systems and Computers, 2005. Conference Record of the Thirty-Ninth Asilomar Conference on*, 28 - november 1, 2005, pp. 1186 – 1190.

[19] F. de Dinechin and B. Pasca, "Floating-point exponential functions for dsp-enabled fpgas," in *Field-Programmable Technology (FPT), 2010 International Conference on*, dec. 2010, pp. 110 –117.

[20] R. G. Lyons, *Understanding Digital Signal Processing (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

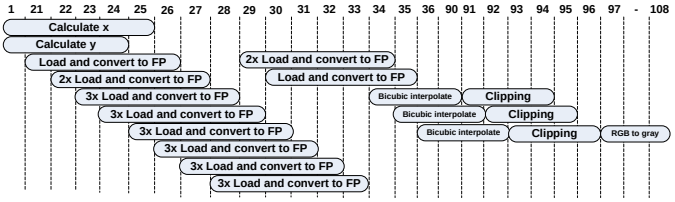
APPENDIX A1 : ANALYSIS OF AUTOESL ACCELERATORS

A global schedule or the code of the accelerators made with AutoESL can be found here. This gives an insight on why some of the pipeline depths are quite large.

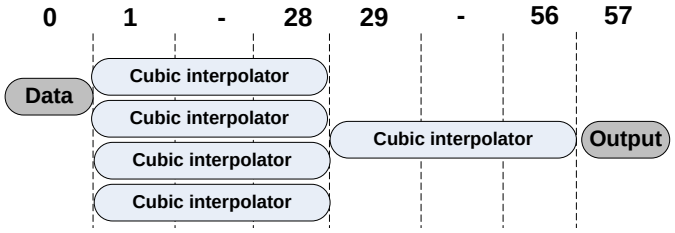
Resize image and RGB to gray conversion accelerator

The image resize and RGB to gray accelerator has a pipeline depth of 108 stages and an initiation interval of 8 cycles. For the bicubic interpolator, which resizes the image, 16 pixels are needed and the BRAM has 2 ports. Therefore, an initiation interval of 8 cycles is needed.

In Fig. 17 the global cycle diagram can be found of the whole accelerator and the bicubic interpolator. Most of the stages are due to calculating the x- and y-coordinate and also because the bicubic interpolation is executed as two 1D steps. Further, most of the area used in the accelerator is due to the bicubic interpolator because there are 4 parallel executions of the cubic interpolator.



(a) Cycle diagram bicubic interpolator and RGB to gray accelerator



(b) Cycle diagram bicubic interpolator

Fig. 17. Cycle diagram bicubic interpolator and RGB to gray accelerator

```
float x, temp_x;
int exp_x;

...
for(j = 0; j < IMG_RES_WIDTH ; j++) {
  /* calculate x */
  temp_x = tx*j;
  exp_x = (int) (temp_x);
  x = (float) exp_x;
  if(x > temp_x) {
   x = x - 1;
   exp_x = exp_x - 1;
  }
  frac_x = temp_x - x;
  for(i = 0; i < IMG_RES_HEIGHT ; i++) {
    /* calculate y */
    ...
  }
}
```

Code Example 3. calculate x

Code 3 shows how the fractional part of a floating point is taken. The variable tx is the scale which is $\frac{img\_width}{64}$. To get the current x-coordinate, tx is multiplied with the loop variable j. Thereafter, the x-coordinate is being rounded by converting it to an integer. If it is rounded up, the variable needs to be subtracted by 1. The fractional part is then the original floating minus the exponent part of the x-coordinate. The same is also done for calculating the y-coordinate.

Average Filter and subtraction accelerator

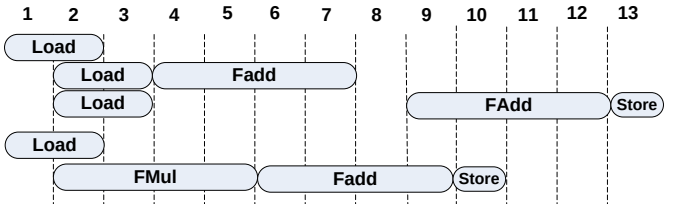
The average filter is executed as two 1D convolution kernels as shown in Eqn. 20. Further, the filter has an initialization part, which fills a temporary BRAM with the results of adding the rows. Thereafter, the main part is started where columns and rows are executed simultaneously.

Fig. 18 shows the cycle diagram for the average filter. The initialization part has an initiation interval of 2 cycles and a pipeline depth of 13 stages as shown in Fig. 18a. Further, the two different equations that are calculated in Eqn. 21 are also in the figure.

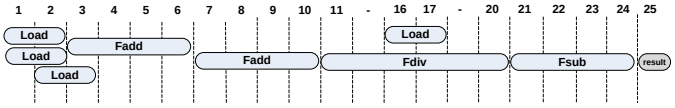
The main part has an initiation interval of 5 cycles and a pipeline depth of 25 stages. Fig. 18b shows the cycle diagram of when the columns are added, divided by 9 and the result is subtracted from the original. The summation of the rows is not drawn in the figure, but it is done in parallel using the same cycle diagram as shown in Fig. 18a.

$$P_{avg}[x,y] = \frac{1}{9}\sum_{j=-1}^{1}\sum_{i=-1}^{1} P[x-i, y-j] \tag{20}$$

$$P_{avg\_1D}[x] = \begin{cases} 2P[x] + P[x+1] & \text{if } x = 0 \\ 2P[x] + P[x-1] & \text{if } x = \text{WIDTH}-1 \\ P[x-1] + P[x] + & \\ P[x+1] & \text{others} \end{cases} \tag{21}$$



(a) Cycle diagram average filter accelerator row



(b) Cycle diagram average filter accelerator column

Fig. 18. Cycle diagram average filter accelerator

Calculate complex number accelerator

Calculating complex number is split into two parts. First, is to calculate the argument of the resized Fourier transformed image using the inverse tangent function. Second part is using the magnitude and the argument to calculate the complex number.

The inverse tangent accelerator has a initiation interval of 1 cycle and a pipeline depth of 46 stages. The schedule of the inverse tangent accelerator is straightforward and follows the algorithm in Code 4 exactly. Therefore, we left out the cycle diagram. The accelerator begins with 3 comparisons and thereafter if needed the pre-calculations for the inverse tangent. Thereafter, the inverse tangent is computed with some post-calculations.

```
float inv_tan(float a)
{
  float result;

  result = a/(1+0.28125f*a*a);

  return result;
}

void arctan(float img[N], float result[N])
{
  int i;
  LOOP1:
  for(i=0;i<N;i++) {
  if(img[i] == PI) result[i] = PI;
  else if(img[i] < -1.0f) result[i] =
      2*(-PI/2 - inv_tan(1.0f/img[i]));
  else if(img[i] > 1.0f) result[i] =
      2*(PI/2 - inv_tan(1.0f/img[i]));
  else result[i] = 2.0f*inv_tan(img[i]);
  }
}
```

Code Example 4. Inverse tangent code

The complex number accelerator uses 2 FSLs, 1 for the magnitude and another for the argument. This results in a initiation interval of 1 cycle and a pipeline depth of 66 stages. The sine and cosine are approximated using a Taylor series of 12th and 13th order respectively. The algorithm can be found in Code 5. A part of the schedule is shown in Fig. 19. From stage 13, it can be seen that the subtractions and additions alternate.

Gaussian Filter accelerator

The Gaussian filter accelerator is implemented in a similar way as the average filter accelerator. It also uses two 1D convolutions, but non-existing pixels are 0. Eqn. 22 shows the equation. The initiation interval of the initialization part is 5 and the pipeline depth is 56 stages. For the main part, the initiation interval is 7 cycles and the pipeline depth is 96 stages.

$$P_{gauss}[x,y] = \sum_{j=-4}^{5} h_1[j] \sum_{i=-4}^{5} P_{IFFT}[x-i, y-j] \cdot h_1[i] \tag{22}$$

```
void sin_taylor(float img[N], float
    img2[N], float result[N], float
    result2[N])
{
  int i,j, input_counter = 0,
      output_counter = 0;
  float input, mag, temp, cos, sin;
  float factorials[12] = {1.0f/2.0f,
      1.0f/6.0f, ... ,1.0f/6227020800.0f};

  LOOP1:
  for(i=0;i<N;i++) {
    input = img[input_counter];
    temp = img[input_counter];
    sin = img[input_counter];

    mag = img2[input_counter];
    cos = 1.0f;
    input_counter++;
    LOOP2:
      for(j=0;j<12;j+=4) {
        temp = temp*input;
        cos = cos - temp*factorials[j];

        temp = temp*input;
        sin = sin - temp*factorials[j+1];

        temp = temp*input;
        cos = cos + temp*factorials[j+2];

        temp = temp*input;
        sin = sin + temp*factorials[j+3];
      }
      result[output_counter] = mag*cos;
      result2[output_counter] = mag*sin;
      output_counter++;
  }
}
```

Code Example 5. Complex number calculation code

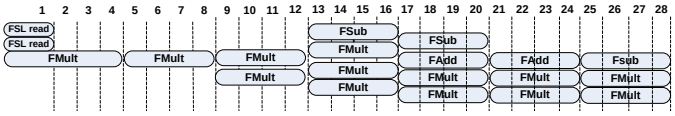


Fig. 19. Partial cycle diagram complex number calculation

Fig. 20 shows a part of the cycle diagram of the Gaussian filter accelerator. The multiply-adds are written in a for loop. Therefore, the next addition can only start when the previous addition is finished. The first add starts in stage 8. In total 9 additions and selections need to be executed and the last select can be done in parallel with the store to BRAM. This results into $45-1$ stages. Further, there are 4 empty stages to avoid the use of multiple adders in parallel. $8+45-1+4=56$, this explains the pipeline depth. The code of the filter is shown in Code 6.

The floating point comparator is hard to explain as there are no floating point comparisons in the algorithm. My guess is that if the pixel needed is outside the image range, the input to floating point multiplier becomes 0. The comparator checks if it 0, and if so it takes the result before the multiplier.

The main part is using a similar schedule in which first the columns are calculated and afterwards the rows. The columns

need 52 stages and then the rows need an additional 44 stages to calculate the result.

```
void gauss_filter(float img[N], float
    result[N])
{
  float temp, buffer[16][WIDTH],
      input[HEIGHT][WIDTH],
      result_gauss[HEIGHT][WIDTH];
  int i,x,y, counter = 0, counter2 = 0;
  int pos_x, pos_y;

  /* copy image */
  for(x=0;x<WIDTH;x++) {
    for(y=0;y<HEIGHT;y++) {
      input[y][x] = img[counter++];
      }
  }

  /* convolute in x-direction */
  for(y = 0; y < 6; y++) {
    for(x = 0; x < WIDTH; x++) {
      temp = 0;

      for(i=-5;i<5;i++) {
        pos_x = x - i;
        if(pos_x >= 0 && pos_x < WIDTH) {
          temp +=
              coeffs[i+5]*input[y][pos_x];
        }
      }
      buffer[y][x] = temp;
    }
  }

  /* convolute in y-direction */
  unsigned int j = 6;
  for(y=0;y<HEIGHT;y++) {
    for(x=0;x<WIDTH;x++) {
      temp = 0;

      for(i=-5;i<5;i++) {
        pos_y = y-i;
        if(pos_y >= 0 && pos_y < HEIGHT)
          temp += coeffs[i+5]*
              buffer[pos_y&0x0F][x];
      }
      result_gauss[y][x] = temp;

      /* then again in x-direction */
      temp = 0;

      for(i=-5;i<5;i++) {
        pos_x = x - i;
        pos_y = y+6;
        if(pos_x >= 0 && pos_x < WIDTH) {
          temp += coeffs[i+5]*
              input[pos_y][pos_x];
        }
      }
      buffer[j][x] = temp;
      if((j+1) > 15) j = 0;
      else j++;
  }
}
```

Code Example 6. Code gauss filter

Finding the minimum and maximum of all the values needs an initiation interval of 2 cycles and a pipeline depth of 5 stages. AutoESL chooses an initiation interval of 2 cycles, because in that way it only needs 1 floating point comparator. Normalizing the values from 0 to 1, has a initiation interval of 1 cycle and a pipeline depth of 11 stages. These two parts are fairly simple. Therefore no cycle diagram is drawn.

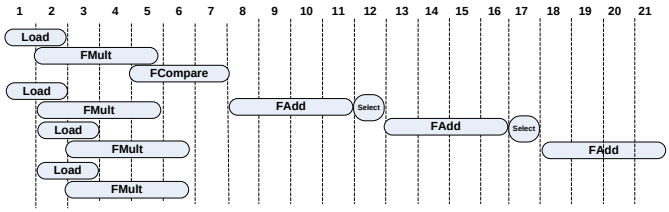


Fig. 20. Partial cycle diagram Gaussian filter

Overall, we conclude that many improvements can be made to this accelerator. First of all, finding the minimum and maximum can be integrated with the convolution loop.

Further, the additions take a lot of stages. Manually balancing the multiply-adds can lead to less stages. But the BRAM is still the bottleneck as it can only load 2 pixels at a time. Using a shift register would improve the execution, but this requires also more floating point units.